# Resource analysis by sup-interpretation

Jean-Yves Marion, Romain Péchoux

# Resource Analysis by Sup-interpretation

Jean-Yves Marion and Romain Péchoux

Loria, Calligramme project, B.P. 239,
54506 Vandœuvre-lès-Nancy Cedex, France
École Nationale Supérieure des Mines de Nancy, INPL, France
{Jean-Yves.Marion, Romain.Pechoux}@loria.fr

**Abstract.** We propose a new method to control memory resources by static analysis. For this, we introduce the notion of sup-interpretation which bounds from above the size of function outputs. We establish a criteria for which the stack frame size is polynomially bounded. The criteria analyses terminating as well as non-terminating programs. This method applies to first order functional programming with pattern matching. This work is related to quasi-interpretations but we are now able to determine resources of different algorithms and it is easier to perform an analysis with this new tools.

## 1 Introduction

This paper deals with general investigation on program complexity analysis. It introduces the notion of sup-interpretation, a new tool that provides an upper bound on the size of every stack frame if the program is non-terminating, and establishes an upper bound on the size of function outputs if the program is terminating.

A sup-interpretation of a program is a *partial* assignment of function symbols, which ranges over reals and which bounds the size of the computed values.

The practical issue is to provide program static analysis in order to guarantee space resources that a program consumes during an execution. There is no need to say that this is crucial for at least many critical applications, and have strong impact in computer security. There are several approaches which are trying to solve the same problem. The first protection mechanism is by monitoring computations. However, if the monitor is compiled with the program, it could crash unpredictably by memory leak. The second is the testing-based approach, which is complementary to static analysis. Indeed, testing provides a lower bound on the memory while static analysis gives an upper bound. The gap between both bounds is of some value in practical applications (See [22] for an interesting discussion). Lastly, the third approach is type checking done by a bytecode verifier. In an untrusted environment (like embedded systems), the type protection policy (Java or .Net) does not allow dynamic allocation. Our approach is an attempt to control resources, and provide a proof certificate, of a high-level language in such a way that the compiled code is safe wrt memory overflow. Thus, we capture and deal with memory allocation features.

M. Hagiya and P. Wadler (Eds.): FLOPS 2006, LNCS 3945, pp. 163–176, 2006.

Similar approaches are the one by Hofmann [15, 16] and the one by Aspinall and Compagnoni [5].

For that purpose we consider first order functional programming language with pattern matching but we firmly believe that such a method could be applied to other languages such as resource bytecode verifier by following the lines of [2], language with synchronous cooperative threads as in [3] or first order functional language including streams as in [14] .

The notion of sup-interpretation can be seen as a kind of annotation provided in the code by the programmer. Sup-interpretations strongly inherit from the notion of quasi-interpretation developed by Bonfante, Marion and Moyen in [10, 11, 21]. Consequently the notion of sup-interpretation comes from the notion of polynomial interpretation used to prove termination of programs in [13, 18] and more recently in [8, 20]. Quasi-interpretation, like sup-interpretation, provides a bound over function outputs by static analysis for first order functional programs and allows the programmer to find a bound on the size of every stack frame. Quasi-interpretation was developed with the aim to pay more attention to the algorithmic aspects of complexity than to the functional (or extensional) one and then it is part of study of the implicit complexity of programs.

However the notions of sup-interpretation and quasi-interpretation differ for two reasons. First, the sup-interpretations are partial assignments which do not satisfy the subterm property, and this allows to capture a larger class of algorithms. In fact, programs computing logarithm or division admits a sup-interpretation but have no quasi-interpretation. Second, the sup-interpretation is a partial assignment over the set of function symbols of a program, whereas the quasi-interpretation is a total assignment on function symbols. On the other hand, sup-interpretations come with a companion, which is a weight to measure argument size of recursive calls involved in a program run. In order to obtain a polynomial space bound, some constraints are developed over weights and sup-interpretations using the underlying notion of dependency pairs by Arts and Giesl [4]. The dependency pairs were initially introduced for proving termination of term rewriting systems automatically. Even if this paper no longer focuses on termination, the notion of dependency pair is used for forcing the program to compute in polynomial space. There is a very strong relation between termination and computational complexity. Indeed, in order to prove some complexity bounds and termination, we need to control the arguments occurring in a function recursive call. Since we try to control together the arguments of a recursive call, the sup-interpretation is closer to the dependency pairs method than to the size-change principle method of [19] which consider the arguments of a recursive call separately (See more rencently [17]). Section 2 introduces the first order functional language and its semantics. Section 3 defines the main notions of sup-interpretation and weight used to bound the size of a program outputs. Section 4 presents the notion of fraternity used to control the size of values added by recursive calls. In section 5, we define the notion of polynomial and additive assignments for sup-interpretations and weights. Finally, section 6 introduces the notion of friendly programs and

the main theorems of this paper providing a polynomial bound on the values computed by friendly programs. The full paper with all proofs is available at `http://www.loria.fr/~pechoux`. The appendix of the full paper presents examples of friendly programs, an example of non-friendly program and some examples over streams.

## 2 First Order Functional Programming

### 2.1 Syntax of Programs

We define a generic first order functional programming language. The vocabulary $\Sigma = \langle Cns, Op, Fct \rangle$ is composed of three disjoint domains of symbols. The arity of a symbol is the number $n$ of arguments that it takes. The set of programs are defined by the following grammar.

$$\begin{array}{lll}
\texttt{Programs} \ni \mathbf{p} & ::= & def_1, \cdots, def_m \\
\texttt{Definitions} \ni def & ::= & \mathtt{f}(x_1, \cdots, x_n) = e^{\mathtt{f}} \\
\texttt{Expression} \ni e & ::= & x \mid \mathbf{c}(e_1, \cdots, e_n) \mid \mathbf{op}(e_1, \cdots, e_n) \mid \mathtt{f}(e_1, \cdots, e_n) \\
 & & \mid \mathbf{Case}\ e_1, \cdots, e_n\ \mathbf{of}\ \overline{p_1} \to e^1 \ldots \overline{p_\ell} \to e^\ell \\
\texttt{Patterns} \ni p & ::= & x \mid \mathbf{c}(p_1, \cdots, p_n)
\end{array}$$

where $\mathbf{c} \in Cns$ is a constructor, $\mathbf{op} \in Op$ is an operator, $\mathtt{f} \in Fct$ is a function symbol, and $\overline{p_i}$ is a sequence of $n$ patterns. Throughout, we generalize this notation to expressions and we write $\overline{e}$ to express a sequence of expressions, that is $\overline{e} = e_1, \ldots, e_n$, for some $n$ clearly determined by the context.

The set of variables $Var$ is disjoint from $\Sigma$ and $x \in Var$. In a definition, $e^{\mathtt{f}}$ is called the body of $\mathtt{f}$. A variable of $e^{\mathtt{f}}$ is either a variable in the parameter list $x_1, \cdots, x_n$ of the definition of $\mathtt{f}$ or a variable which occurs in a pattern of a case definition. In a case expression, patterns are not overlapping. The program's main function symbol is the first function symbol in the program's list of definitions. We usually don't make the distinction between this main symbol and the program symbol $\mathbf{p}$.

Lastly, it is convenient, because it avoids tedious details, to restrict case definitions in such a way that an expression involved in a **Case** expression does not contain nested **Case** (In other words, an expression $e^j$ does not contain an expression **Case** ). This is not a severe restriction since a program involving nested **Case** can be transformed in linear time in its size into an equivalent program without the nested case construction.

### 2.2 Semantics

The set $Values$ is the constructor algebra freely generated from $Cns$.

$$Values \ni v ::= \mathbf{c} \mid \mathbf{c}(v_1, \cdots, v_n) \qquad \mathbf{c} \in Cns$$

Put $Values^* = Values \cup \{\mathbf{Err}\}$ where **Err** is the value associated to an error. Each operator **op** of arity $n$ is interpreted by a function $[\![\mathbf{op}]\!]$ from $Values^n$

to $Values^*$. Operators are essentially basic partial functions like destructors or characteristic functions of predicates like $=$. The destructor **tl** illustrates the purpose of **Err** when it satisfies $[\![\mathbf{tl}(\mathbf{nil})]\!] = \mathbf{Err}$.

The computational domain is $Values^{\#} = Values \cup \{\mathbf{Err}, \bot\}$ where $\bot$ means that a program is non-terminating. The language has a closure-based call-by-value semantics which is displayed in Figure 1. A few comments are necessary. A substitution $\sigma$ is a finite function from variables to $Values$. The application of a substitution $\sigma$ to an expression $e$ is noted $e\sigma$.

$$\frac{t_1 \downarrow w_1 \dots t_n \downarrow w_n}{\mathbf{c}(t_1, \cdots, t_n) \downarrow \mathbf{c}(w_1, \cdots, w_n)}\ \mathbf{c} \in Cns \text{ and } \forall i, w_i \neq \mathbf{Err}$$

$$\frac{t_1 \downarrow w_1 \dots t_n \downarrow w_n}{\mathbf{op}(t_1, \cdots, t_n) \downarrow [\![\mathbf{op}]\!](w_1, \cdots, w_n)}\ \mathbf{op} \in Op \text{ and } \forall i, w_i \neq \mathbf{Err}$$

$$\frac{e \downarrow u \quad \exists \sigma,\ i\ :\ p_i\sigma = u \quad e_i\sigma \downarrow w}{\mathbf{Case}\ e\ \mathbf{of}\ p_1 \to e_1 \dots p_\ell \to e_\ell \downarrow w}\ \mathbf{Case} \text{ and } u \neq \mathbf{Err}$$

$$\frac{e_1 \downarrow w_1 \dots e_n \downarrow w_n \quad \mathtt{f}(x_1, \cdots, x_n) = e^{\mathtt{f}} \quad e^{\mathtt{f}}\sigma \downarrow w}{\mathtt{f}(e_1, \cdots, e_n) \downarrow w}\ \text{where } \sigma(x_i) = w_i \neq \mathbf{Err} \text{ and } w \neq \mathbf{Err}$$

**Fig. 1.** Call by value semantics of ground expressions wrt a program **p**

The meaning of $e \downarrow w$ is that $e$ evaluates to the value $w$ of $Values^*$. If no rule is applicable, then an error occurs, and $e \downarrow \mathbf{Err}$. So, a program **p** computes a partial function $[\![\mathbf{p}]\!] : Values^n \to Values^{\#}$ defined as follows. For all $v_i \in Values$, $[\![\mathbf{p}]\!](v_1, \cdots, v_n) = w$ iff $\mathbf{p}(v_1, \cdots, v_n) \downarrow w$. Otherwise $[\![\mathbf{p}]\!](v_1, \cdots, v_n) = \bot$. Throughout, we shall say that $[\![\mathbf{p}]\!](v_1, \cdots, v_n)$ is defined when $[\![\mathbf{p}]\!](v_1, \cdots, v_n)$ is a constructor term of $Values$.

## 3 Sup-interpretations

### 3.1 Partial Assignment

A partial assignment $\theta$ is a partial mapping from a vocabulary $\Sigma$ such that for each symbol $\mathtt{f}$ of arity $n$, in the domain of $\theta$, it yields a partial function $\theta(f) : (\mathbb{R})^n \longmapsto \mathbb{R}$. The domain of a partial assignment $\theta$ is noted $\mathrm{dom}(\theta)$. Because it is convenient, we shall always assume that partial assignments that we consider, are defined on constructors and operators. That is $Cns \cup Op \subseteq \mathrm{dom}(\theta)$.

An expression $e$ is defined over $\mathrm{dom}(\theta)$ if each symbol belongs to $\mathrm{dom}(\theta)$ or is a variable of *Var*. Take a denumerable sequence $X_1, \ldots, X_n, \ldots$. Assume that an expression $e$ is defined over $\mathrm{dom}(\theta)$ and has $n$ variables. The partial assignment of $e$ wrt $\theta$ is the extension of the assignment $\theta$ to the expression $e$ that we write $\theta^*(e)$. It denotes a function from $\mathbb{R}^n$ to $\mathbb{R}$ and is defined as follows:

1. If $x_i$ is a variable of *Var*, let $\theta^*(x_i) = X_i$
2. If $b$ is a 0-ary symbol of $\Sigma$, then $\theta^*(b) = \theta(b)$.
3. If $\overline{e}$ is a sequence of $n$ expressions, then $\theta^*(\overline{e}) = \max(\theta^*(e_1), \ldots, \theta^*(e_n))$
4. If $e$ is a **Case** expression of the shape **Case** $\overline{e}$ **of** $\overline{p_1} \to e^1 \ldots \overline{p_\ell} \to e^\ell$,

$$\theta^*(e) = \max(\theta^*(\overline{e}), \theta^*(e^1), \ldots, \theta^*(e^\ell))$$

5. If $f$ is a symbol of arity $n > 0$ and $e_1, \cdots, e_n$ are expressions, then

$$\theta^*(f(e_1, \cdots, e_n)) = \theta(f)(\theta^*(e_1), \ldots, \theta^*(e_n))$$

### 3.2 Sup-interpretation

**Definition 1 (Sup-interpretation).** *A sup-interpretation is a partial assignment $\theta$ which verifies the three conditions below :*

1. *The assignment $\theta$ is weakly monotonic. That is, for each symbol $f \in dom(\theta)$, the function $\theta(f)$ satisfies*

$$\forall i = 1, \ldots, n \; X_i \geq Y_i \Rightarrow \theta(f)(X_1, \cdots, X_n) \geq \theta(f)(Y_1, \cdots, Y_n)$$

2. *For each $v \in Values$,*

$$\theta^*(v) \geq |v|$$

   *The size of an expression $e$ is noted $|e|$ and is defined by $|\mathbf{c}| = 0$ where $\mathbf{c}$ is a 0-ary symbol and $|\mathbf{b}(e_1, \ldots, e_n)| = 1 + \sum_i |e_i|$ where $\mathbf{b}$ is a $n$-ary symbol.*
3. *For each symbol $f \in dom(\theta)$ of arity $n$ and for each value $v_1, \ldots, v_n$ of Values, if $[\![f]\!](v_1, \ldots, v_n)$ is defined, that is $[\![f]\!](v_1, \ldots, v_n) \in Values$, then*

$$\theta^*(f(v_1, \ldots, v_n)) \geq \theta^*([\![f]\!](v_1, \ldots, v_n))$$

Now an expression $e$ admits a sup-interpretation $\theta$ if $e$ is defined over $\mathrm{dom}(\theta)$. The sup-interpretation of $e$ wrt $\theta$ is $\theta^*(e)$.

Intuitively, the sup-interpretation is a special program interpretation. Instead of yielding the program denotation, a sup-interpretation provides an approximation from above of the size of the outputs of the function denoted by the program. It is worth noticing that sup-interpretations are a complexity measure in the sense of Blum [7].

**Lemma 1.** *Let $e$ be an expression with no variable and which admits a sup-interpretation $\theta$. Assume that $[\![e]\!]$ is defined. We then have*

$$\theta^*([\![e]\!]) \leq \theta^*(e)$$

*Proof.* The proof is done by structural induction on expression. The base case is a consequence of Condition 2 of Definition 1.

Take an expression $e = f(e_1, \cdots, e_n)$ that has a sup-interpretation $\theta$. By induction hypothesis (IH), we have $\theta^*(e_i) \geq \theta^*([\![e_i]\!])$. Now,

$$\begin{aligned}
\theta^*(e) &= \theta(f)(\theta^*(e_1), ..., \theta^*(e_n)) && \text{by definition of } \theta^* \\
&\geq \theta(f)(\theta^*([\![e_1]\!]), ..., \theta^*([\![e_n]\!])) && \text{by 1 of Dfn 1 and (IH)} \\
&= \theta^*(f([\![e_1]\!], ..., [\![e_n]\!])) && \text{by definition of } \theta^* \\
&\geq \theta^*([\![f]\!]([\![e_1]\!], ..., [\![e_n]\!])) && \text{by 3 of Dfn 1} \\
&= \theta^*([\![e]\!])
\end{aligned}$$

□

Given an expression $e$, we define $\|e\|$ thus:

$$\|e\| = \begin{cases} |[\![e]\!]| & \text{if } [\![e]\!] \text{ is defined} \\ 0 & \text{otherwise} \end{cases}$$

**Corollary 1.** *Let $e$ be an expression with no variable and which admits a sup-interpretation $\theta$. Assume that $[\![e]\!]$ is defined. We then have*

$$\|e\| \leq \theta^*(e)$$

*Proof.*

$$\begin{aligned}
\theta^*(e) &\geq \theta^*([\![e]\!]) && \text{by Lemma 1} \\
&\geq \|e\| && \text{by Condition 2 of Dfn 1}
\end{aligned}$$

□

*Example 1.*

$$\begin{aligned}
\mathtt{half}(x) \ = \mathbf{Case}\ x\ \mathbf{of}\ & \mathbf{0} \to \mathbf{0} \\
& \mathbf{S}(\mathbf{0}) \to \mathbf{0} \\
& \mathbf{S}(\mathbf{S}(y)) \to \mathbf{S}(\mathtt{half}(y))
\end{aligned}$$

In this example, the function `half` computes $\lfloor n/2 \rfloor$ on an entry of size $n$. So by taking $\theta(\mathbf{S})(X) = X+1$ and $\theta(\mathtt{half})(X) = X/2$, we define a sup-interpretation of the function symbol `half`. In fact, both functions are monotonic. For every unary value $v$ of size $n$, $\theta^*(v) = n \geq n = |v|$ by definition of $\theta(\mathbf{S})$, so that condition 2 on sup-interpretation is satisfied. Finally, it remains to check that for every value $v$, $\theta^*(\mathtt{half}(v)) \geq \theta^*([\![\mathtt{half}(v)]\!])$. For a value $v$ of size $n$, we have by definition of $\theta^*$ that $\theta^*(\mathtt{half}(v)) = \theta^*(v)/2 = n/2$ and $\theta^*([\![\mathtt{half}(v)]\!]) = \|\mathtt{half}(v)\| = \lfloor n/2 \rfloor$. Since $n/2 \geq \lfloor n/2 \rfloor$, condition 3 of sup-interpretation is satisfied. Notice that such a sup-interpretation is not a quasi-interpretation (a fortiori not an interpretation for proof termination) since it does not have the subterm property (see below for a definition of this property).

### 3.3 Weight

The weight allows us to control the size of the arguments in recursive calls. A weight is an assignment having the subterm property but no longer giving a

bound on the size of a value computed by a function. Intuitively, whereas the sup-interpretation controls the size of the computed values, the weight can be seen as a control point for the computation of recursive calls.

**Definition 2 (Weight).** *A weight $\omega$ is a partial assignment which ranges over Fct. To a given function symbol $\mathtt{f}$ of arity $n$ it assigns a total function $\omega_{\mathtt{f}}$ from $\mathbb{R}^n$ to $\mathbb{R}$ which satisfies:*

1. *$\omega_{\mathtt{f}}$ is weakly monotonic.*

$$\forall i = 1, \ldots, n,\ X_i \geq Y_i \Rightarrow \omega_{\mathtt{f}}(\ldots, X_i, \ldots) \geq \omega_{\mathtt{f}}(\ldots, Y_i, \ldots)$$

2. *$\omega_{\mathtt{f}}$ has the subterm property*

$$\forall i = 1, \ldots, n,\ \omega_{\mathtt{f}}(\ldots, X_i, \ldots) \geq X_i$$

The weight of a function is often taken to be the maximum or the sum functions.

The monotonicity property combined with the fact that a weight ranges over function symbols ensures suitable properties on the number of occurrences of a loop in a program when we consider the constraints given in section 6. Moreover, the subterm property allows to control the size of each argument in a recursive call, in opposition to the size-change principle as mentioned in the introduction.

## 4 Fraternities

In this section we define fraternities which are an important notion based on dependency pairs, that Arts and Giesl [4] introduced to prove termination automatically. Fraternities allow to tame the size of arguments of recursive calls.

A *context* is an expression $\mathsf{C}[\diamond_1, \cdots, \diamond_r]$ containing one occurrence of each $\diamond_i$. Here, we suppose that the $\diamond_i$'s are new symbols which are not in $\Sigma$ nor in *Var*. The substitution of each $\diamond_i$ by an expression $d_i$ is noted $\mathsf{C}[d_1, \cdots, d_r]$.

**Definition 3.** *Assume that $\mathtt{f}(x_1, \cdots, x_n) = e^{\mathtt{f}}$ is a definition of a program.* An expression $d$ is activated *by $\mathtt{f}(p_1, \cdots, p_n)$ where the $p_i$'s are patterns if there is a context with one hole $\mathsf{C}[\diamond]$ such that:*

- *If $e^{\mathtt{f}}$ is a compositional expression (that is with no case definition inside it), then $e^{\mathtt{f}} = \mathsf{C}[d]$. In this case, $p_1 = x_1 \ldots p_n = x_n$.*
- *Otherwise, $e^{\mathtt{f}} =$ **Case** $e_1, \cdots, e_n$ **of** $\overline{q_1} \rightarrow e^1 \ldots \overline{q_\ell} \rightarrow e^\ell$, then there is a position $j$ such that $e^j = \mathsf{C}[d]$. In this case, $p_1 = q_{j,1} \ldots p_n = q_{j,n}$ where $\overline{q_j} = q_{j,1} \ldots q_{j,n}$.*

At first glance, this definition may look a bit tedious. However, it is convenient in order to predict the computational data flow involved. Indeed, an expression is activated by $\mathtt{f}(p_1, \cdots, p_n)$ when $\mathtt{f}(v_1, \cdots, v_n)$ is called and each $v_i$ matches the corresponding pattern $p_i$.

The notion of activated expression provides a precedence $\geq_{Fct}$ on function symbols. Indeed, set $\mathtt{f} \geq_{Fct} \mathtt{g}$ if there are $\overline{e}$ and $\overline{p}$ such that $\mathtt{g}(\overline{e})$ is activated

by $\mathtt{f}(\overline{p})$. Then, take the reflexive and transitive closure of $\geq_{Fct}$, that we also note $\geq_{Fct}$. It is not difficult to establish that $\geq_{Fct}$ is a preorder. Next, say that $\mathtt{f} \approx_{Fct} \mathtt{g}$ if $\mathtt{f} \geq_{Fct} \mathtt{g}$ and inversely $\mathtt{g} \geq_{Fct} \mathtt{f}$. Lastly, $\mathtt{f} >_{Fct} \mathtt{g}$ if $\mathtt{f} \geq_{Fct} \mathtt{g}$ and $\mathtt{g} \geq_{Fct} \mathtt{f}$ does not hold.

Intuitively, $\mathtt{f} \geq_{Fct} \mathtt{g}$ means that $\mathtt{f}$ calls $\mathtt{g}$ in some executions. And $\mathtt{f} \approx_{Fct} \mathtt{g}$ means that $\mathtt{f}$ and $\mathtt{g}$ call themselves recursively.

Say that an expression $d$ activated by $\mathtt{f}(p_1, \cdots, p_n)$ is maximal if there is no context $\mathsf{C}[\diamond]$, distinct from the empty context, such that $\mathsf{C}[d]$ is activated by $\mathtt{f}(p_1, \cdots, p_n)$.

**Definition 4.** *In a program* $\boldsymbol{p}$*, an expression* $d = \mathsf{C}[\boldsymbol{g}_1(\overline{e_1}), \ldots, \boldsymbol{g}_r(\overline{e_r})]$ *activated by* $\boldsymbol{f}(p_1, \cdots, p_n)$ *is a fraternity if*

1. *d is maximal*
2. *For each* $i \in \{1, r\}$, $\boldsymbol{g}_i \approx_{Fct} \boldsymbol{f}$.
3. *For every function symbol* $\boldsymbol{h}$ *that appears in the context* $\mathsf{C}[\diamond_1, \cdots, \diamond_r]$*, we have* $\boldsymbol{f} >_{Fct} \boldsymbol{h}$.

All along, we suppose that there is no nested fraternities, which means that a fraternity $d$ does not contain any fraternity inside it. This restriction prevents definitions of the shape $\mathtt{f}(\mathbf{S}(x)) = \mathtt{f}(\mathtt{f}(x))$. This restriction is not too strong since such functions are not that natural in a programming perspective and either they have to be really restricted or they rapidly generate complex functions like the Ackermann one. The following examples illustrate typical fraternity constructions.

*Example 2.* Consider the program $\mathtt{log}$ computing $\mathtt{log}_2(n) + 1$ on an entry of size $n$ and using the program $\mathtt{half}$ of example 1.

$$\begin{aligned}
\mathtt{log}(x) &= \mathbf{Case}\ x\ \mathbf{of}\ \mathbf{0} \to \mathbf{0} \\
&\qquad\qquad \mathbf{S}(y) \to \mathbf{S}(\mathtt{log}(\mathtt{half}(\mathbf{S}(y)))) \\
\mathtt{half}(x) &= \mathbf{Case}\ x\ \mathbf{of}\ \mathbf{0} \to \mathbf{0} \\
&\qquad\qquad \mathbf{S}(\mathbf{0}) \to \mathbf{0} \\
&\qquad\qquad \mathbf{S}(\mathbf{S}(y)) \to \mathbf{S}(\mathtt{half}(y))
\end{aligned}$$

This program admits two fraternities $\mathbf{S}(\mathtt{log}[\mathtt{half}(\mathbf{S}(y))])$ and $\mathbf{S}[\mathtt{half}(y)]$ since $\mathtt{log} >_{Fct} \mathtt{half}$. Take $\mathbf{S}(\mathtt{log}[\mathtt{half}(\mathbf{S}(y))])$, this fraternity is decomposed into a context $\mathbf{S}(\mathtt{log}[\diamond])$ and an expression $\mathtt{half}(\mathbf{S}(y))$.

*Example 3 (division).* Consider the following definitions that encode the division $\lceil n/m \rceil$ on two entries of sizes $n$ and $m > 0$:

$$\begin{aligned}
\mathtt{minus}(x, y) = \mathbf{Case}\ x, y\ \mathbf{of}\ \mathbf{0}, z &\to \mathbf{0} \\
\mathbf{S}(z), \mathbf{0} &\to \mathbf{S}(z) \\
\mathbf{S}(u), \mathbf{S}(v) &\to \mathtt{minus}(u, v)
\end{aligned}$$

$$\mathtt{q}(x,y) = \mathbf{Case}\ x,y\ \mathbf{of}\ \mathbf{0},\mathbf{S}(z) \to \mathbf{0}$$
$$\mathbf{S}(z),\mathbf{S}(u) \to \mathbf{S}(\mathtt{q}(\mathtt{minus}(z,u),\mathbf{S}(u)))$$

This program admits two fraternities $\mathtt{minus}(u,v)$ and $\mathbf{S}[\mathtt{q}(\mathtt{minus}(z,u),\mathbf{S}(u))]$ since $\mathtt{q} >_{Fct} \mathtt{minus}$.

**Definition 5.** *A* state *is a tuple $\langle \mathbf{f}, u_1, \cdots, u_n\rangle$ where $\mathbf{f}$ is a function symbol of arity $n$ and $u_1, \ldots, u_n$ are values. Assume that $\eta_1 = \langle \mathbf{f}, u_1, \cdots, u_n\rangle$ and $\eta_2 = \langle \mathbf{g}, v_1, \cdots, v_k\rangle$ are two states. Assume also that $\mathsf{C}[\mathbf{g}(e_1, \cdots, e_k)]$ is activated by $\mathbf{f}(p_1, \cdots, p_n)$. A* transition *is a triplet $\eta_1 \overset{\mathsf{C}[\diamond]}{\rightsquigarrow} \eta_2$ such that:*

1. *There is a substitution $\sigma$ such that $p_i\sigma = u_i$ for $i = 1, \ldots, n$,*
2. *and $[\![e_j\sigma]\!] = v_j$ for $j = 1...k$.*

*We call such a graph a call-tree of $\mathbf{f}$ over values $u_1, \ldots, u_n$ if $\langle \mathbf{f}, u_1, \cdots, u_n\rangle$ is its root. A state may be seen as a stack frame. A call-tree of root $\langle \mathbf{f}, u_1, \cdots, u_n\rangle$ represents all the stack frames which will be pushed on the stack when we compute $\mathbf{f}(u_1, \ldots, u_n)$.*

## 5 Polynomial Assignments

**Definition 6.** *A partial assignment $\theta$ is polynomial if for each symbol $f$ of arity $n$ of $dom(\theta)$, $\theta(f)$ is* ***bounded*** *by a polynomial of $\mathbb{R}[X_1, \cdots, X_n]$. A polynomial sup-interpretation is a polynomial assignment. A polynomial weight $\omega$ of arity $n$ is a weight which is bounded by some polynomial of $\mathbb{R}[X_1, \cdots, X_n]$.*

An assignment of $\mathbf{c} \in \mathrm{dom}(\theta)$ is *additive* if

$$\theta(\mathbf{c})(X_1, \cdots, X_n) = \sum_{i=1}^{n} X_i + \alpha_{\mathbf{c}} \qquad \alpha_{\mathbf{c}} \geq 1$$

If the polynomial assignment of each constructor is additive then the assignment is additive. Throughout the following paper we consider additive assignments. As a consequence we have the following lemma:

**Lemma 2.** *There is a constant $\alpha$ such that for each value $v$ of Values, the inequality is satisfied :*

$$|v| \leq \theta^*(v) \leq \alpha|v|$$

## 6 Local Criteria to Control Space Resources

**Definition 7 (Friendly).** *A program $\boldsymbol{p}$ is* friendly *iff there is a polynomial sup-interpretation $\theta$ and a polynomial weight $\omega$ such that for each fraternity expression $d = \mathsf{C}[\mathbf{g}_1(\overline{e_1}), \ldots, \mathbf{g}_r(\overline{e_r})]$ activated by $\mathbf{f}(p_1, \cdots, p_n)$ we have,*

$$\theta^*(\mathsf{C}[\diamond_1,\ldots,\diamond_r]) = \max_{i=1..r}(\diamond_i + R_i(Y_1,\ldots,Y_m))$$

*with $R_i$ polynomials where each $Y_i$ corresponds to a variable occurring in C.*
*Moreover, for each $i \in \{1, r\}$, we have that for each substitution $\sigma$,*

$$\omega_f(\theta^*(p_1\sigma),\ldots,\theta^*(p_n\sigma)) \geq \omega_{g_i}(\theta^*(e_{i,1}\sigma),\ldots,\theta^*(e_{i,m}\sigma))$$

*Moreover, if*

$$\exists\sigma \quad \omega_f(\theta^*(p_1\sigma),\ldots,\theta^*(p_n\sigma)) = \omega_{g_i}(\theta^*(e_{i,1}\sigma),\ldots,\theta^*(e_{i,m}\sigma))$$

*Then $R_i(Y_1,\ldots,Y_m)$ is the null polynomial.*

*Example 4.* The program of example 2 is friendly. We take $\theta(\mathbf{S})(X) = X + 1$ and $\theta(\mathtt{half})(X) = X/2$. The contexts of the two fraternities involved in this program are $\mathbf{S}[\diamond]$, thus having a sup-interpretation $\theta^*(\mathbf{S}[\diamond]) = \diamond + 1$. We have to find $\omega_{\mathtt{log}}$ and $\omega_{\mathtt{half}}$ such that for every $\sigma$:

$$\omega_{\mathtt{log}}(\theta^*(\mathbf{S}(y\sigma))) > \omega_{\mathtt{log}}(\theta^*(\mathtt{half}(\mathbf{S}(y\sigma))))$$
$$\omega_{\mathtt{half}}(\theta^*(\mathbf{S}(\mathbf{S}(y\sigma)))) > \omega_{\mathtt{half}}(\theta^*(y\sigma))$$

Both inequalities are satisfied by taking $\omega_{\mathtt{log}}(X) = \omega_{\mathtt{half}}(X) = X$. Thus the program is friendly.

*Example 5.* The program of example 3 is friendly by taking $\theta(\mathbf{S})(X) = X + 1$, $\theta(\mathtt{minus})(X, Y) = X$, $\omega_{\mathtt{minus}}(X, Y) = \max(X, Y)$ and $\omega_{\mathtt{q}}(X, Y) = X + Y$. An example of an unfriendly program is given in the full paper.

**Theorem 1.** *Assume that $\boldsymbol{p}$ is a friendly program. For each function symbol $\boldsymbol{f}$ of $\boldsymbol{p}$ there is a polynomial $P$ such that for every value $v_1,\ldots,v_n$,*

$$\|\boldsymbol{f}(v_1,\ldots,v_n)\| \leq P(\max(|v_1|,...,|v_n|))$$

*Proof.* The proof can be found in the full paper. It begins by assigning a polynomial $P_{\mathtt{f}}$ to every function symbol $\mathtt{f}$ of a friendly program. This polynomial is the sum of a bound on the size of values added by the contexts of recursive calls and of a bound on the size of values added by the calls which are no longer recursive. Then it checks both bounds thus showing that the values computed by the program are polynomially bounded. □

The programs presented in examples 2 and 3 are examples of friendly programs and thus computing polynomially bounded values. More examples of friendly programs can be found in the appendix of the full paper.

The next result strengthens Theorem above. Indeed it claims that even if a program is not terminating then the intermediate values are polynomially bounded. This is quite interesting because non-terminating process are common, and moreover it is not difficult to introduce streams with a slight modification of the above Theorem, which is essentially based on the semantics change.

**Theorem 2.** *Assume that* $\boldsymbol{p}$ *is a friendly program. For each function symbol* $\boldsymbol{f}$ *of* $\boldsymbol{p}$ *there is a polynomial* $R$ *such that for every node* $\langle \boldsymbol{g}, u_1, \cdots, u_m \rangle$ *of the call-tree of root* $\langle \boldsymbol{f}, v_1, \cdots, v_n \rangle$,

$$\max_{j=1..m} (|u_j|) \leq R(\max(|v_1|, ..., |v_n|))$$

*even if* $\boldsymbol{f}(v_1, \ldots, v_n)$ *is not defined.*

*Proof.* The proof is in the full paper and is a consequence of previous theorem since in every state of the call-tree, the values are computed and thus bounded polynomially. □

*Remark 1.* As mentioned above, this theorem holds for non-terminating programs and particularly for a class of programs including streams. For that purpose we have to give a new definition of substitutions over streams. In fact, it would be meaningless to consider a substitution over stream variables. Thus stream variables are never substituted and the sup-interpretation of a stream $l$ is taken to be a new variable $L$ as in the definition of the sup-interpretations.

*Example 6 (Streams).* Let e::l be a stream with :: a stream constructor symbol, e an expression (the head of the stream) and l a stream variable (the tail of the stram) and suppose that we have already defined a semantics over streams in a classical way.

$$\texttt{addstream}(x, y) = \textbf{Case } x, y \textbf{ of } z :: l, u :: l' \rightarrow \texttt{add}(z, u) :: \texttt{addstream}(l, l')$$

Then this (merging) program is friendly by taking $\theta^*(l) = L$, $\theta(\texttt{add})(X, Y) = X + Y$, $\theta^*(x :: l) = \theta^*(x) + L + 1$ and $\omega_{\texttt{addstream}}(X, Y) = X + Y$. Thus a variant of theorem 2 holds. The variation comes from the fact that it would be non-sense to consider streams as inputs, since the size of a stream is unbounded. Consequently, the inputs are chosen to be a restricted number of stream heads. In the same way, every mapping program over streams of the shape:

$$\texttt{f}(x) = \textbf{Case } x \textbf{ of } z :: l \rightarrow \texttt{g}(z) :: \texttt{f}(l)$$

is friendly if g represents a friendly program. Thus the variant of theorem 2 also applies. Moreover for all these programs we know that the values computed in the output streams (i.e. in the heads of right-hand side definition) are polynomially bounded in the size of some of the inputs (heads) since the computations involve only friendly functions over non-stream datas (else some parts of the program will never be evaluated). Finally an example of non-friendly program is:

$$\texttt{f}(x) = \textbf{Case } x \textbf{ of } z :: l \rightarrow \texttt{f}(z :: z :: l)$$

In fact, this program does not fit our requirements since it adds infinitely the head of the stream to its argument, computing thus an unbounded value.

## 7 Conclusion and Perspectives

The notion of sup-interpretation allows to check that the size of the outputs of a friendly program is bounded polynomially by the size of it inputs. It allows to capture algorithms admitting no quasi-interpretations (division, logarithm, gcd ...). So, our experiments show that is not too difficult to find sup-interpretations for the following reasons. First, we have to guess sup-interpretations and weights of only some, and not all, symbols. Second, a quasi-interpretations for those symbol works pretty well in most of the cases. And so we can use tools to synthesize quasi-interpretations [1, 9]. Our works is related to semi-automatic procedure to analyse complexity, see for example Benzinger [6] for programs extracted from Nuprl or Debray et al. for logic programs [12].

Sup-interpretation should be easier to synthesize than quasi-interpretations since we have to find fewer assignments. Moreover it is not so hard to find a sup-interpretation, since quasi-interpretation often defines a sup-interpretation, except in the case of additive contexts. Indeed, consider the program $\mathtt{f}(\mathbf{c}(x, y)) = \mathbf{c}(\mathtt{f}(x), \mathtt{f}(y))$ defined over binary trees. It admits the identity function as quasi-interpretation. However it does not admit a sup-interpretation since its context should have an additive sup-interpretation. Which is clearly impossible in the friendly criteria. Hopefully this drawback only relies on the friendly criteria and not on the sup-interpretation itself. We are currently working on a more general criteria which should be able to capture such programs and whose aim is to capture entirely the class of programs admitting a quasi-interpretation. As a consequence every quasi-interpretation satisfying the friendly criteria is a sup-interpretation. A programmer is also interested in bounding the total number of stack frames. This issue is partially tackled in the full paper by showing that cycles of the call-tree corresponding to friendly programs with strict inequalities in the friendly criteria have a number of occurrences bounded by a polynomial in the size of the inputs. Since every occurence of a cycle is composed of a bounded number of states (which depends directly on the size of the program) and since every cycle occurs a polynomial number of times, we know that the corresponding call-tree will have a number of states bounded polynomially in the size of the inputs. Consequently, for such programs, both stack frame sizes and the number of stack-frames are polynomially bounded by the size of the inputs. The corresponding termination result strongly inherits in a natural way from the dependency pairs method of Arts and Giesl [4]. However, it differs in the sense that the monotonicity of the quasi-ordering and the inequalities over definitions (rules) of a program are replaced by the notion of sup-interpretation combined to weights. Consequently, it shares the same advantages and disadvantages than the dependency pairs method compared to termination methods such as size-change principle by Jones et al. [19], failing on programs admitting no polynomial orderings (Ackermann function, for example), and managing to prove termination on programs where the size-change principle fails. For a more detailed comparison between both termination criteria see [23]. Finally, an open question concerns characterization of time complexity classes with the use of such a tool, particularly, the characterization of polynomial time by determining a restriction on sup-interpretations.

## References


1. R. Amadio. Max-plus quasi-interpretations. In Martin Hofmann, editor, *Typed Lambda Calculi and Applications, 6th International Conference, TLCA 2003, Valencia, Spain, June 10-12, 2003, Proceedings*, volume 2701 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2003.
2. R. Amadio, S. Coupet-Grimal, S. Dal-Zilio, and L. Jakubiec. A functional scenario for bytecode verification of resource bounds. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic, 18th International Workshop, CSL 13th Annual Conference of the EACSL, Karpacz, Poland*, volume 3210 of *Lecture Notes in Computer Science*, pages 265–279. Springer, 2004.
3. R. Amadio and S. Dal Zilio. Resource control for synchronous cooperative threads. Research Report LIF.
4. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
5. D. Aspinall and A. Compagnoni. Heap bounded assembly language. *Journal of Automated Reasoning (Special Issue on Proof-Carrying Code)*, 31:261–302, 2003.
6. R. Benzinger. Automated higher-order complexity analysis. *Theoretical Computer Science*, 318(1-2):79–103, 2004.
7. M. Blum. A machine-independent theory of the complexity of recursive functions. *Journal of the Association for Computing Machinery*, 14:322–336, 1967.
8. G. Bonfante, A. Cichon, J.-Y. Marion, and H. Touzet. Algorithms with polynomial interpretation termination proof. *Journal of Functional Programming*, 11(1):33–53, 2001.
9. G. Bonfante, J.-Y. Moyen J.-Y. Marion, and R. Péchoux. Synthesis of quasi-interpretations. *Workshop on Logic and Complexity in Computer Science, LCC2005, Chicago*, 2005. `http://www.loria/~pechoux`.
10. G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. On lexicographic termination ordering with space bound certifications. In Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin, editors, *Perspectives of System Informatics, 4th International Andrei Ershov Memorial Conference, PSI 2001, Akademgorodok, Novosibirsk, Russia, Ershov Memorial Conference*, volume 2244 of *Lecture Notes in Computer Science*. Springer, Jul 2001.
11. G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. Quasi-interpretation a way to control resources. *Submitted to Theoretical Computer Science*, 2005. `http://www.loria.fr/~moyen/appsemTCS.ps`.
12. S.K. Debray and N.W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, 1993.
13. N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, pages 69–115, 1987.
14. S.G. Frankau and A. Mycroft. Stream processing hardware from functional language specifications. In Martin Hofmann, editor, *36th Hawai'i International Conference on System Sciences (HICSS 36)*. IEEE, 2003.
15. M. Hofmann. Linear types and non-size-increasing polynomial time computation. In *Proceedings of the Fourteenth IEEE Symposium on Logic in Computer Science (LICS'99)*, pages 464–473, 1999.
16. M. Hofmann. A type system for bounded space and functional in-place update. In *European Symposium on Programming, ESOP'00*, volume 1782 of *Lecture Notes in Computer Science*, pages 165–179, 2000.

17. A. Hugh and S.C. Khoo. Affined-based size-change termination. *APLAS 2003, Beijing*, 2003.
18. D.S. Lankford. On proving term rewriting systems are noetherien. Technical report, 1979.
19. Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Symposium on Principles of Programming Languages*, volume 28, pages 81–92. ACM press, january 2001.
20. S. Lucas. Polynomials over the reals in proofs of termination: from theory to practice. *RAIRO Theoretical Informatics and Applications*, 39(3):547–586, 2005.
21. J.-Y. Marion and J.-Y. Moyen. Efficient first order functional program interpreter with time bound certifications. In Michel Parigot and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France*, volume 1955 of *Lecture Notes in Computer Science*, pages 25–42. Springer, Nov 2000.
22. J. Regehr. Say no to stack overflow. 2004. `http://www.embedded.com`.
23. R. Thiemann and J. Giesl. Size-change termination for term rewriting. In *14th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science, Valencia, Spain, 2003. Springer.